RICE UNIVERSITY

Flow-Controlled Background Replication for Big Data Jobs

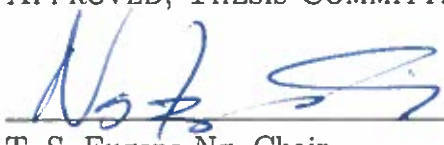
by

Simbarashe Joseph Dzinamarira

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

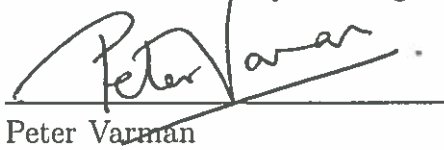
Master of Science

Approved, Thesis Committee:



T. S. Eugene Ng, Chair
Associate Professor of Computer Science
and Electrical and Computer Engineering

Alan Cox
Professor of Computer Science and
Electrical and Computer Engineering



Peter Varman
Professor of Electrical and Computer
Engineering and Computer Science

Houston, Texas

March, 2016

ABSTRACT

Flow-Controlled Background Replication for Big Data Jobs

by

Simbarashe Joseph Dzinamarira

This thesis proposes mechanisms to reduce the cost of data replication. Data replication is an extremely expensive but crucial operation in distributed file systems (DFSs). Replication secures data against system failures but slows down applications by increasing I/O contention in the system. DFSs lie at the foundation of big data processing systems. Therefore, improving them benefits nearly the entire big data ecosystem.

This thesis proposes flow-controlled background replication as a method to minimize the impact of replication on the performance of applications. The proposed system accelerates jobs, exploits under-utilized storage I/O bandwidth, and supports job-based and replica-based bandwidth allocation.

Our implementation, called Pfimbi, improved the runtime of data-intensive jobs by up to 30%. Pfimbi accelerated the job runtimes, in a workload based on a Facebook trace, by 15% on average. Pfimbi successfully improves application runtime while obtaining a work span comparable to that of the common synchronous replication scheme.

Acknowledgements

I would first like to thanks my advisor, Prof T. S. Eugene Ng, for his mentorship and guidance throughout this project. He has taught me how to boldly take ownership of my research and push it forward. This has helped me both in my research and personal life. I would also like to thank Florin Dinu, who has collaborated closely with my advisor and I on this project. He helped me get a handle around the Hadoop source code and how the system works.

Many thanks to my committee members Prof. Alan Cox and Prof. Peter Varman. Their questions and feedback have been very helpful in improving this work and guiding my future research. Thank you Kimberly Macellaro for proofreading this thesis. I would also like to thank the members of my lab, Fred Liu, Ruiqi Liu, Xin Huang, Yiting Xia, and Xiaoye Sun, for their support and friendship along the way, and for their helpful feedback throughout this project.

Lastly, I would like to my parents and siblings for encouraging me in my graduate studies.

Contents

Illustrations

Tables

Chapter 1

Introduction

The Hadoop Distributed File System (HDFS) [1] is the foundation of nearly every big data solution available today. For years, the distributed system's community has been constantly looking to develop new big data processing tools to fit an increasing number of use cases and performance requirements. In this fast-changing landscape, HDFS has remained the preeminent distributed storage solution. Figures 1.1 and 1.2 illustrates the Hadoop [2] and the Spark [3] ecosystems, the most popular big data software stacks today. Underlying both is HDFS.

Given the ubiquity of HDFS, it is critical to ensure good performance for HDFS accesses. The performance of HDFS writes, in particular, have been at the forefront of recent efforts from the industry and the open source community [4, 5].

A fundamental feature of HDFS writes is that data is streamed synchronously through a set of DataNodes that are organized in a pipeline. A copy of the data is created on each of the DataNodes in the pipeline. One effort [4] aimed at clusters using heterogeneous SSD and hard disk storage allows a job to place a subset of its data copies on faster SSDs. This approach is unable to benefit fully from the faster SSDs because the performance of the entire synchronous pipeline is dictated by the slowest portion of the pipeline, usually the much slower hard disks. Another effort [5] allows a DataNode first to store data in a small RAM Disk, then lazily persist the data to non-volatile storage. This lazy persist method, however, does not scale since it has little benefit for jobs that write much more data than the small RAM Disk can

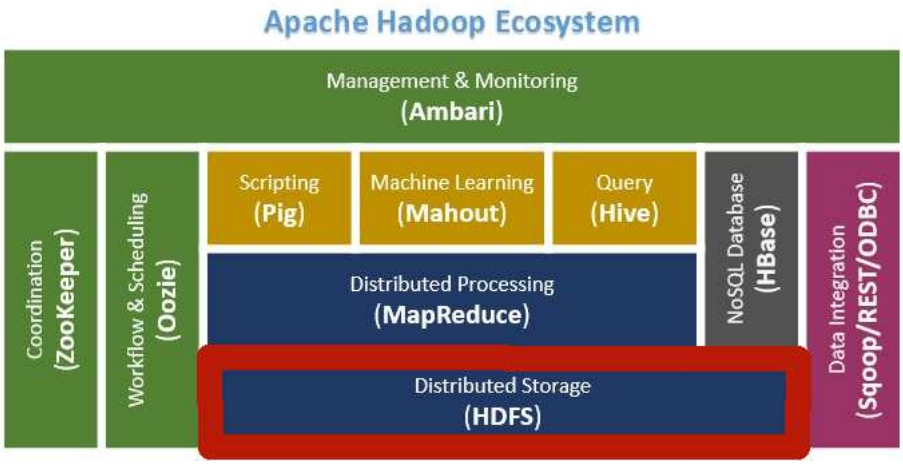


Figure 1.1 : The Hadoop Ecosystem.
Source: `http://www.mssqltips.com/tipimages2/3260_Apache_Hadoop_Ecosystem.JPG`

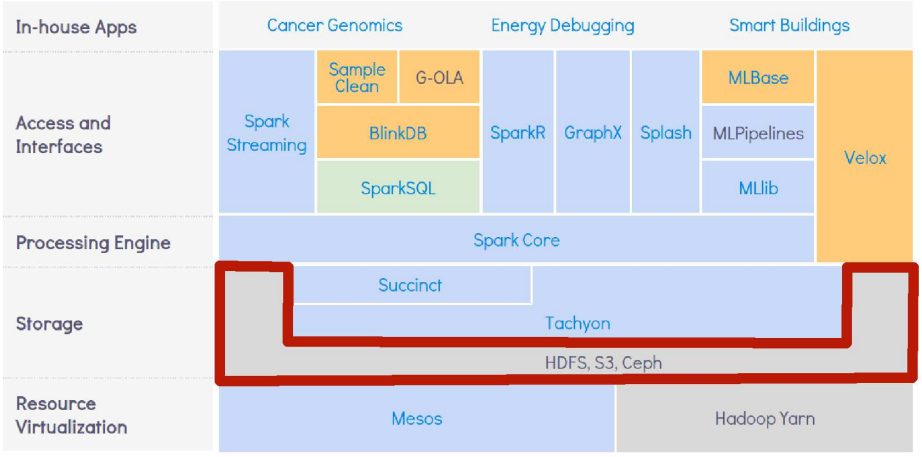


Figure 1.2 : The Spark Ecosystem.
Source: `https://amplab.cs.berkeley.edu/software/`

hold. When the data size is large, performance is ultimately dictated by the slowest portion of the synchronous pipeline.

An alternative to consider would be background replication schemes to eliminate

the synchronous write pipeline by decoupling the primary copy's write from the replica copies' writes. In this way, a job completes as soon as the primary copy has been written, and the job's lifespan is not explicitly tied to the speed of replication. The potential performance benefit is large.

However, our thesis is that this potentially large benefit of background replication is not realizable by simply decoupling primary writes from replication; the key missing ingredient is that all data traffic must be subjected to explicit flow control. Without flow-control background replication writes may still contend arbitrarily with primary writes of executing jobs and among themselves, harming storage I/O efficiency and negating the benefit of background replication.

This paper presents Pfimbi, the first alternative to HDFS that supports flow-controlled background replication. Pfimbi's design fulfils the following important goals:

1. Isolate primary writes from the interference caused by replication writes. Thus, Pfimbi effectively speeds up jobs.

2. Efficiently leverage storage I/O under-utilization periods to perform replication writes. Thus, Pfimbi can sometimes complete replication writes faster than HDFS.

3. Flexibly divide storage I/O bandwidth between jobs. Thus, Pfimbi can selectively accelerate data replication for jobs that need a small data loss vulnerability period.

4. Flexibly divide storage I/O bandwidth between replicas. Thus, Pfimbi can achieve progressively increasing levels of failure resilience for a job's output.

5. Simultaneously support synchronous and background replication. Thus, Pfimbi is backward compatible with jobs that are designed for the original HDFS.

To isolate interference and leverage storage I/O under-utilization, Pfimbi exploits the observation that in real workloads storage I/O under-utilization is plentiful but individual intervals of under-utilization are often brief and interleaved with periods of peak utilization. Moreover, no correlation exists between these periods across DataNodes. Therefore, to ensure good performance data blocks must be transmitted from upstream nodes to a sink node promptly in order to enable the sink node to take advantage of moments of storage I/O inactivity. On the other hand, such transmissions cannot be overly aggressive or they might overwhelm the sink node and eliminate its ability to fully control usage of storage I/O bandwidth. These requirements suggest that upstream and downstream nodes must coordinate their actions through a lightweight protocol. Pfimbi's approach is to use a combination of receiver-side buffering and credit-based flow control.

Pfimbi must be job-aware so it can flexibly divide under-utilized storage I/O bandwidth as well as achieve increasing levels of failure resilience. Each upstream node sources data blocks from multiple tasks belonging to different jobs. Each block belongs to a different replication pipeline, and each block occupies a different logical position within the pipeline. A DataNode in Pfimbi is thus required to manage the transfer of data blocks at this detailed level. Each node in Pfimbi schedules network transfers for replication data using weighted fair queuing (WFQ). Pfimbi also combines WFQ for inter-job scheduling with priority queuing for intra-job scheduling to achieve progressively increasing levels of failure resilience.

We demonstrate that for a job trace derived from a Facebook workload, Pfimbi improves the job runtime by 15% on average, up to 300% for small jobs (writing 1GB),

and up to 30% for the most data-intensive jobs (writing 80GB). When compared to background replication alone, Pfimbi's flow control mechanisms improve job runtime by up to 33%.

Chapter 2

Background and Related Work

We implement Pfimbi by extending the HDFS codebase. Here review how HDFS replicates data.

2.1 Terminology

For a job to complete, its output needs to be written once. We call this the primary copy (primary write). Data replication reproduced the primary copy, creating replicas. For example, when a job requires resilience against two node failures it means that it needs three data copies (a primary and two replicas). A client is application code that reads and writes to the file system. By synchronous replication we mean replication is on the critical path of client writes. The client write will not complete until replication is done. We use the terms background or asynchronous replication to refer to data replication that is decoupled from client writes.

2.2 HDFS architecture

HDFS is used to distribute data over a cluster composed of commodity computer nodes. It uses a master-slave architecture. The master (called NameNode) handles metadata, answers client queries regarding data locations, and directs clients to write data to suitable locations. The slave nodes (called DataNodes) handle the actual client read and write requests. Data is read and written at the granularity of blocks which are typically tens to hundreds of MB in size (commonly 64MB or 256MB). Clusters often collocate storage with computation. The same set of nodes that run compute tasks also run HDFS.

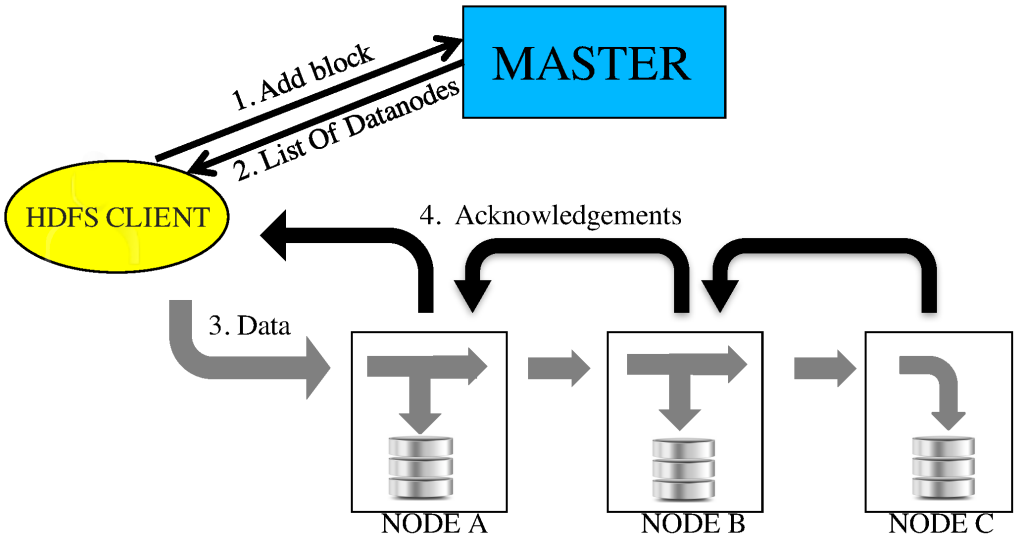


Figure 2.1 : Synchronous pipelined write process in HDFS

2.3 Synchronous pipelined writes in HDFS

HDFS writes blocks in a synchronous pipelined fashion. Figure 2.1 details a synchronous pipelined block write. Every block goes through these steps. First the client contacts the master (1.) and receives a list of nodes that will host the block copies (2.). The size of the list depends on the replication factor of the file (i.e, number of copies). The nodes on the list are chosen such that their locations fulfill a predefined failure-resilience policy (e.g, survive two node failures or one rack failure). Commonly, the first node in the list is the same node that executes the task writing the data. The client then organizes the provided nodes into a pipeline. To improve performance the nodes are ordered to minimize the total network distance from the client to the last node in the pipeline [1].

Once the pipeline is setup up, HDFS sends the block's data over the pipeline at the granularity of application-level packets (3.). When a node receives a packet from

upstream, it forwards the packet downstream and writes the data locally. The last node in the pipeline generates an acknowledgment once it has successfully received the packet (4.). This acknowledgment is propagated upstream, all the way to the client. A window-based scheme is used to limit the maximum number of unacknowledged packets. A packet is considered successfully written after the client receives the corresponding acknowledgment. A block is considered successfully written after the client receives acknowledgments for all packets. This approach ensures that all replicas are made before a write operation to a data block completes. If a failure strikes the data's source node before all replicas in the pipeline are fully written, the data block is considered lost. When the replication factor is large, writing all the replicas can take a considerable amount of time.

2.4 Related Work

Providing resilience against failures is an essential part of any distributed system. However, providing resilience often involves a trade off between cost and performance. This section reviews the different ways to provide resilience, and efforts to reduce the cost and performance degradation caused by these methods. In file systems, resilience is usually provided using replication, erasure coding, or recomputation.

Replication creates several copies of each piece of data and places them on different servers in the system. Data can be replicated in several ways that vary temporally and spatially. Replicating data while writing the data is known as eager replication. A primary write is considered complete only after creating all of the replicas. The alternative is called lazy replication. In lazy replication, replicas can be created soon after the primary write, on a periodic schedule, or in an opportunistic fashion.

Pfimbi and TidyFS [6] are examples of distributed file systems that perform lazy replication. Pfimbi replicates in an opportunistic fashion while TidyFS initiates replication on a periodic schedule. The benefit of replicating data lazily is that our system decouples the performance of an application from replication. A slowdown in replication will not slow the primary write. However, if applications read data soon after writing it, then lazy replication may reduce available copies and result in lower read throughput. The authors of the TidyFS paper found that in their deployment at Microsoft, applications read less than 1% of the data within the first minute of its creation. No more than 10% of the data was read within the first hour of creation. Because the applications in their workload do not usually read data soon after its creation, there was enough time to replicate the data lazily and increase its availability before it is required. These findings mean we also did not need to worry much about low data availability when using Pfimbi.

TidyFS fundamentally differs from what we do in that there is no management of background replication traffic. Once a background replication thread finds pending replicas, it starts creating them immediately regardless of the system load. HDFS can be made to replicate data lazily in a similar fashion using a command called "setRep". We evaluate the performance of such replication in Section 5.3. Because system load is not considered in TidyFS or HDFS using setRep, local client writes will experience contention from background replication traffic leading to poor performance (as Section 5.3 shows). In contrast, Pfimbi uses flow control techniques to replicate data during periods of disk under-utilization. We aim to minimize interference between replication and the primary writes.

Flow control is used in several contexts to share limited capacity between competing flows. One use of flow control in distributed systems is in Retro [7], a resource management framework for multi-tenant distributed systems. Retro can be used to share a set of different resources between tenants. Examples of resources are CPUs, disks, thread pools, and locks. In the case of HDFS, Retro can be used to enforce a given bandwidth sharing policy between jobs belonging to different tenants. In contrast to our asynchronous approach, Retro couples replication primary writes in keeping with the eager replication strategy of HDFS. The result is that replication will continue to compete with and thereby slow down primary writes.

Regardless of whether replication is eager or lazy, techniques could be applied to reduce its cost. Chowdhury et. al developed a system called Sinbad [8] that exploits freedom in choosing the placement of replicas to improve the speed of replication. Sinbad chooses the endpoints of replication pipelines in a way that avoids congested links in the network. Although it works alongside HDFS, which does eager replication, Sinbad could potentially be used to complement a system such as ours that

replicates data lazily. However, with lazy replication, the network information that drives endpoint selection may be stale by the time a node forwards data for replication. We may be able to resolve this by binding replication endpoints at a later time. In our future work, we propose implementing such late binding in Pfimbi. All in all, we view Sinbad as complementary to our work since it reduces network congestion while we reduce disk contention.

So far, we have looked at the ways DFSs create replicas in pipelines: eager vs. lazy replication and adjusting replication endpoints. However, pipelines are not the only way to create replicas. In their work * [9], Islam et. al implemented and evaluated parallel replication streams sharing a single source. Parallel replication reduces the latency of multiple hops in a pipeline. However, parallel replication can quickly be bottlenecked by the egress bandwidth at the source.

Additionally, parallel replication does not address the overhead due to contention between replication and foreground data. The overhead of I/O contention either in the network or on the disk can have a much larger effect on client throughput than latency.

While the works above all use replication for failure resilience, there are works pursuing other strategies. Hitchhiker [10] is one among many systems that use erasure coding for failure resilience. Similar to replication, an erasure-code based system keeps redundant information that can be used to recover lost data. However, rather than keep a duplicate of the original data, an erasure-coded system keeps r parity units for each set of k blocks. This allows for any r of these *(k+r)* to be recovered as long as k units remain available. Erasure codes have a smaller storage footprint than

* "Can Parallel Replication benefit Hadoop Distributed File System for High Performance Interconnects"

replication but lack replication's inexpensive recovery of lost data. If a block is lost in an erasure coded system, a lot of data has to be transferred and must computation must be performed to recover the lost block.

Lastly, resilience can be provided by tracking the lineage of data and recomputing it when lost. Spark [3], Tachyon [11], and RCMP [12] by default choose not to perform replication. If data is lost, these systems re-run the minimum set of tasks necessary to regenerate the lost data. However, under failures, the re-computation process could take as long as many previous tasks need to be re-computed. To bound the amount of re-computation that may need to be done after a failure, these systems can periodically replicate their data as well. When periodically replicating data, it is desirable to do so as efficiently as possible.

Chapter 3

Motivation

This section provides the motivation for Pfimbi. First, we discuss the numerous drawbacks of synchronous replication. Second, we explain why asynchronous replication is safe for most replication writes. Third, we discuss why consistent replicas can be obtained conveniently without synchronous replication. Finally, we discuss why exploitable storage I/O under-utilization may frequently exist.

3.1 Drawbacks of synchronous replication

Synchronous replication couples replication with primary writes, which puts replication on the critical path of the primary writes. This design has negative implications for both performance and resource efficiency.

Synchronous replication causes contention between primary writes and replica writes even within a single job. This contention leads to disproportionally increased job completion time. Take sorting 1TB of data on 30 nodes in Hadoop for example. We found that adding one replica causes a slowdown of 23%, but disproportionally, adding two replicas causes a slowdown of 65%.

Synchronous replication also leads to inefficient cluster-wide resource usage. Since replication exacerbates task execution times, tasks hold on to their allocated resources (CPU, memory) for longer than necessary. By allowing other tasks to use these resources, the system could increase cluster-wide throughput.

Slow DataNodes (e.,. caused by a temporary overload) greatly compound the problems described. Since one node can serve multiple replication pipelines simultaneously, one single slow node can delay several tasks simultaneously. Finally, synchronous replication increases a task's exposure to network congestion. Any slow network transfer can slow down the task.

3.2 Asynchronous replication is safe for most use cases

Most replication writes are performed only for failure resilience and not for performance gain. The mean time between failures (days for a small cluster [13] and hours for a large cluster [14]) is considerably larger than any delay caused by asynchronous background replication. Thus, the failure resilience of jobs will not be negatively impacted very much by asynchronous replication.

Only a small fraction of replication writes can bring performance benefits via improved data locality. Large jobs are responsible for the vast majority of writes [15] in a cluster. Because these large jobs have many tasks, their blocks are naturally spread over the cluster even without replication. Therefore, future jobs reading this data can easily be scheduled for data locality. Small jobs that write little data may indeed benefit from synchronous replication to spread the data over a larger portion of the cluster before a next job reads the data. However, small jobs are responsible for only a minority of the writes [15] so even if their replication is performed synchronously (as it should be), the impact is minimal.

3.3 Consistent replicas without synchronous replication

When data parallel jobs choose among several copies of data, they rely on the fact that all the copies are in sync. The benefit of synchronous replication is that in a general

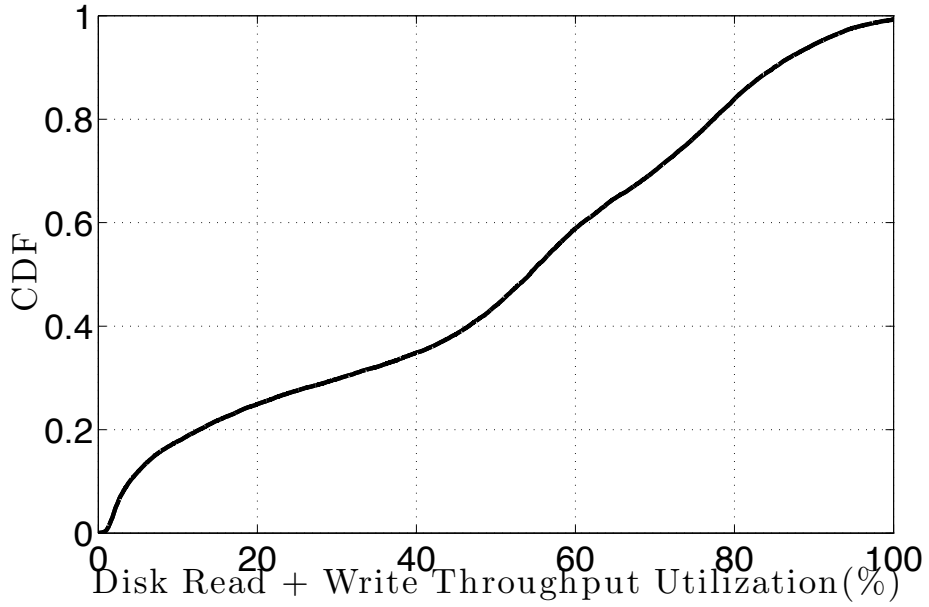


Figure 3.1 : CDF of the proportion of disk bandwidth utilized for all nodes

mix of read and write operations on a file, it guarantees consistency between the copies at the end of a write operation. However, in the case of the write-once read-many file access model, a file system does not fundamentally require synchronous replication. Instead, the same guarantees may be obtained from the file system master node without any impediment on scalability. By default, the master needs to be notified when a replica of a data block is completed on a DataNode. Therefore, the master can guarantee that no incomplete replica will be ever made visible to clients.

The write-once read-many file access model is very popular. HDFS supported only the write-once read-many model from 2007 to August 2010. Since August 2010, HDFS has had limited capability for file updates in the form of the append operation. The append operation was introduced for the specific case of running Apache HBase, a database, over HDFS. The write-once read-many model continues to be the common

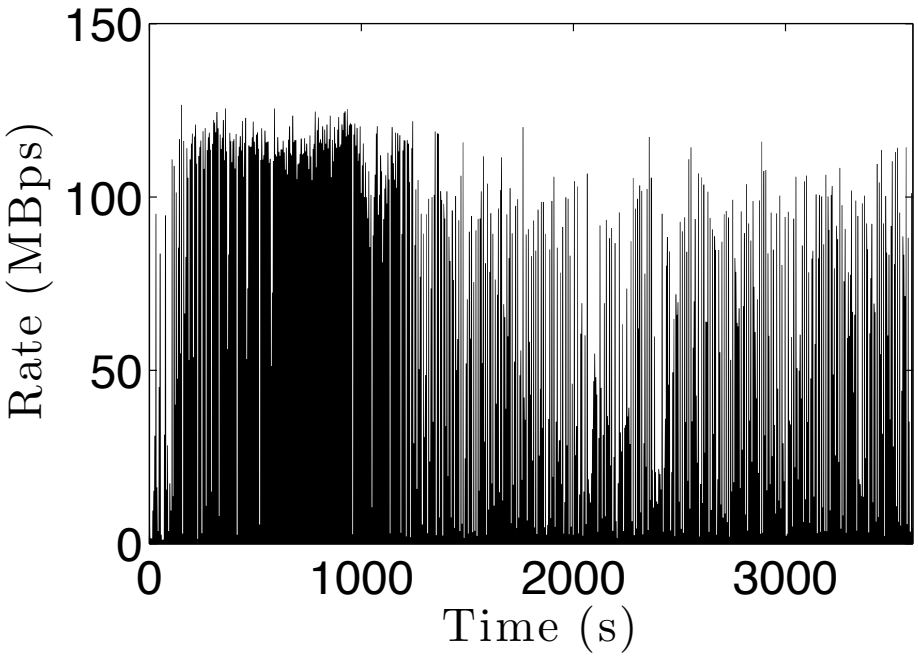


Figure 3.2 : Magnitude of disk bandwidth utilized on one node

use case.

3.4 Exploitable storage I/O under-utilization

Plentiful but irregular storage I/O under-utilization can stem from the fundamental properties of the jobs. Different jobs have different dominant resources and thus put different pressure on the storage subsystem. Some jobs are storage I/O bound (e.g, Terasort [16], NutchIndexing [17], and Bayesian Classification [18]) while others are CPU bound (e.g, Kmeans Clustering [19]). Even a single task may use the storage subsystem differently throughout its lifetime. For example, when configured with enough memory a Hadoop reducer is more storage I/O bound during the write phase compared to the shuffle phase.

To visualize this storage I/O under-utilization, we ran the SWIM workload injector [20] with the first 100 jobs of a 2009 Facebook trace provided with SWIM. We computed the average disk utilization (reads + writes) by periodically reading OS disk activity counters every 100ms. Every 10s we logged a computed sample. Each node had one 2TB HDD used solely by the workload. Figure 3.1 shows the CDF of disk bandwidth usage for all 30 nodes in our cluster. Figure 3.2 shows the timeline of disk bandwidth usage on one node.

Figure 3.1 shows that disk bandwidth was underutilized. Such underutilization provides an opportunity to complete data replication in the background in a timely fashion. Figure 3.2 shows the irregular pattern of disk activity. Periods of idleness or reduced activity were frequently interleaved with periods of peak activity. This observation suggests the need for a solution for managing background replication that can rapidly leverage even short periods of under-utilization.

Chapter 4

System Design and Implementation

4.1 Pfimbi supports both synchronous and asynchronous replication

Pfimbi recognizes that different jobs have different needs and allows jobs to retain the choice of replicating data synchronously. Thus, Pfimbi is backward compatible with jobs designed for the original HDFS. As described in Section 3.2 this can benefit short jobs by ensuring better data locality for the jobs processing their output. Nevertheless, Pfimbi is specifically designed to perform efficient background replication.

4.2 Flow control to minimize interference and leverage disk under-utilization

Pfimbi is designing to meet two goals: minimize the interference caused by replication; and rapidly and efficiently utilize periods of disk under-utilization to perform background replication writes. To achieve these goals, Pfimbi uses a combination of intra-node and inter-node flow-control.

4.2.1 Each node control when replication writes are pushed to the disk

To achieve both goals requires a node that performs background replication must be in full control of the timing of the corresponding disk writes. The replication writes should only be triggered by disk under-utilization and not by any external factors. In this respect, synchronous pipelined replication is inadequate because it does not allow full control. It forces a node to write data to disk as soon as the data is received even though the disk may be fully utilized. In Pfimbi a node retains full control over replication writes. It decides when to perform replication writes by periodically monitoring disk activity. When the disk activity is lower than a disk

activity threshold, nodes start writing background data to disk.

4.2.2 Disk activity and threshold setting

Pfimbi uses I/O request latency as an indicator of disk activity. I/O requests to the disk have a larger latency when the disk is busy. Initially, we chose to use disk throughput as a measure of disk activity. However, when multiple clients are writing concurrently, throughput may be low yet the disk will be busy. Latency captures time spent seeking, it correctly reports high disk activity even as concurrent writes degrade throughput.

An effective threshold must be automatically learned and dynamic, for several reason. First, different nodes may have different latencies so one global threshold will not suit all nodes. Node variation may be attributed to differences in disk technologies, such as SSDs vs. HDDs. Pfimbi can learn a suitable threshold independently for each node, thereby allowing the scheme to work well with heterogeneous hardware. Second, a dynamic threshold can adapt to changes in disk performance. For example, for HDDs, the latency changes as the disk fills up, or when it becomes fragmented. Therefore, a static threshold would mislead the scheme when such changes occur.

4.2.3 Inter-node flow control

Periods of disk under-utilization are often interleaved with periods of peak utilization (see Section 3.4). Therefore, a node has to react fast when disk under-utilization is detected. Rapidly leveraging disk under-utilization periods requires that data be already present at a node, readily available to be written to disk. It would be inefficient to trigger remote network reads at the moment when disk under-utilization is detected because of potentially long transfer times. Thus, blocks requiring asyn-

chronous replication are first buffered at nodes in an in-memory FIFO queue before they are written to disk. Synchronous blocks go directly to disk as in plain HDFS.

The use of the buffers coupled with control over the timing of the writes requires that the node must have control over when the data is received. Otherwise, the buffers could overflow, and the node would be forced to either flush the data to disk or discard it. To let downstream nodes controls the flow of data, Pfimbi uses a credit-based flow control scheme. A node asks other nodes for replication data when there is enough room in its in-memory buffer. The goal is to keep the buffer at high occupancy such that we can leverage any under-utilization period. An additional benefit of this mechanism is that nodes can choose from which upstream node to received data from, thus prioritizing certain transfers.

Figure 4.1 showcases the use of flow control for replication in Pfimbi in an example that assumes two replicas are written asynchronously. The client contacts the master (1.) and receives (2.) a list of 3 nodes (A, B, C), each of which will hold a copy of the block. Node A receives the block (3.), writes it to disk(4.) and sends an acknowledgement (5.). At this point, the client's write operation is complete. Then, A informs B that a new replication block is available for it (6.). Since B has room in its buffer, it immediately requests the block (7.) and starts receiving it (8.). Node B waits until the disk is under-utilized and writes the block to disk (9.). Then it notifies Node C that a block is available (10.). Node C has its buffer full so it cannot receive new blocks (11.). Node C waits until it writes buffered blocks to disk to free up buffer space, and then requests the block from B (12.). Finally, Node C receives the block (13.) and will write it to disk when the disk is under-utilized.

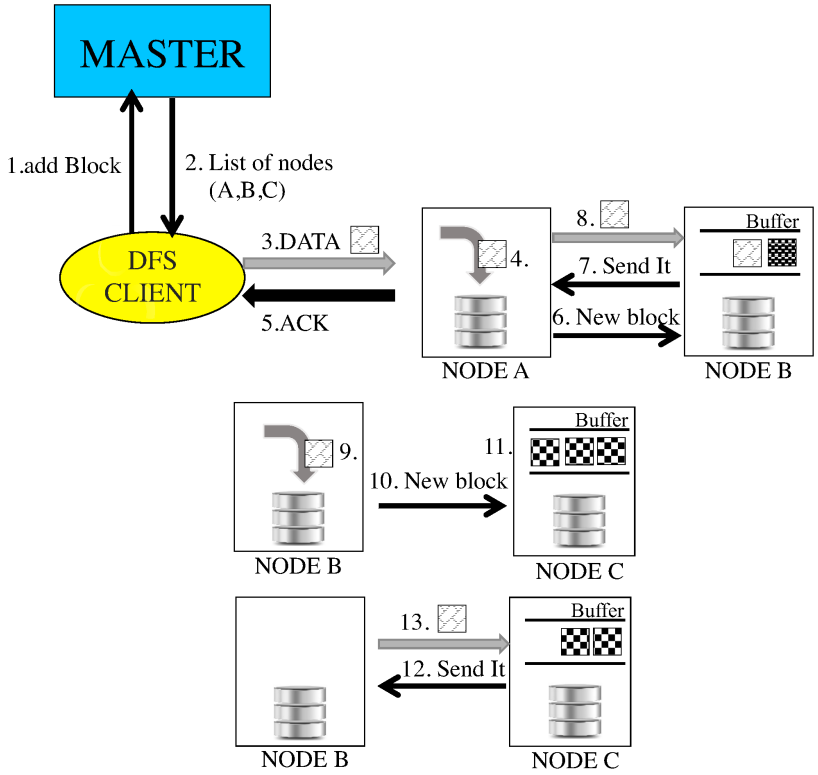


Figure 4.1 : Flow control in Pfimbi

4.2.4 Bandwidth sharing for background replication

For many workloads, enough idleness exists for pending replication work to be performed. However, some I/O intensive workloads keep the disk busy for extended intervals. The continuous high activity would result in replication being throttled indefinitely. To address this case, Pfimbi allows a share of the bandwidth to be guaranteed for background replication by using a weighted round robin approach. After receiving a predefined number of foreground blocks, Pfimbi can flush a background block regardless of the current activity to ensure background replication is not starved.

4.3 Flexible storage I/O bandwidth distribution

A node is typically part of multiple pipelines at the same time. These pipelines can originate from different clients of the same job (e.g, different reducer tasks) or clients from multiple jobs. Moreover, for any of these pipelines, the node can be in different positions (e.g, first, second, etc.). Thus, the node has to solve a scheduling problem and choose which upstream node to ask for a new block. When faced with this choice, Pfimbi addresses our last two goals: to share under-utilized disk bandwidth between jobs and achieve progressively increasing levels of failure resilience for a job.

In Pfimbi, block scheduling is done with Weighted Fair Queuing (WFQ). The blocks from one job comprise a separate flow. Pfimbi assigns the blocks in each flow to a different queue. Each flow has a weight. The weight of a flow determines the amount of bandwidth that flow will get. We assume that weights are assigned via an out-of-band mechanism. The queue whose head block has the earliest estimated completion time is allowed to initiate the reception of its head block. The estimated completion times are calculated using the principles of WFQ.

In Pfimbi, reducing the time to failure resilience for a job is accomplished by giving that job a larger weight. Pfimbi achieves progressively increasing levels of failure resilience by giving priority, within each queue, to the block writes for earlier positions in the pipeline.

Achieving progressively increasing levels of failure resilience is desirable because it improves failure resilience. Without Pfimbi, the blocks to be replicated are treated by nodes indiscriminately, regardless of the node's position in the pipeline of that block. Such treatment is detrimental to failure resilience. Assume the goal is to write three copies of each block. Because of indiscriminate treatment, the writes for position two in the pipeline will likely complete at the same time as the writes for

position three. Assume this process takes 10 seconds. We instead suggest services blocks from position two first such that resilience against one node failure is achieved after 5 seconds. The resilience against two node failures is achieved after the same time (10 seconds), but resilience against one node failure is achieved much faster (5 instead of 10 seconds).

4.4 Failure handling

If the node hosting the client (i.e, the task that writes) fails, then the pipelined write stops. Typically, the same node hosting the client also hosts the primary copy. This failure case falls outside of the scope of a DFS and is typically treated by the big data framework with existing task-restart mechanisms.

If the node that fails is within the synchronous segment of the pipeline, but not the same node hosting the client, then the pipelined write also stops. In this case the client times out the write and selects another pipeline to retry. Existing DFSs behave in a similar way.

If a node in the asynchronous part of the pipeline fails, then the pipeline will be severed at that node. All synchronous or asynchronous copies upstream of the failed node can complete normally. The node immediately upstream of the failed node also continues normally. If this upstream node sends a block notification to the failed node, then it will receive and error and stop retrying. If the block notification was sent before the failure, but there is no block request arriving from the failed node, the upstream node is still unaffected. This is because the upstream node does not maintain any extra state after sending the block notification since the downstream node drives the remaining data transfer. If no further action is taken, the data will remain under-replicated. To obtain the desired number of replicas, Pfimbi relies

on the block-loss recovery mechanisms already supported by the master node. The master node periodically checks for under-replicated blocks within the file system and starts their replication in an out-of-band manner until the desired number of replicas is reached.

4.5 Scalability

Existing DFSs that underlie big data frameworks have been proven to be highly scalable. Pfimbi is just as scalable as existing DFSs because its overall file system architecture is the same. Importantly, Pfimbi's centralized master performs the exact set and number of operations, keeping the load on the master remains. Pfimbi also retains the pipelined design to replication. Pfimbi only changes the manner in which the pipeline is managed. Finally, the coordination mechanism Pfimbi introduces to enable flow control is lightweight and local to a pair of upstream-downstream nodes. Thus, the coordination mechanisms in Pfimbi scale with the size of the DFS.

4.6 Design alternatives

4.6.1 Activity metric

Our first design choice was to use disk latency as a metric for disk activity. However, whenever data is being drained for the buffer cache, disk latency will be high, so Pfimbi throttles replication. This is the correct behavior when more foreground data is flowing into the buffer cache. When no data is being written from the user space into the buffer cache, and the OS is simply draining the buffer cache, Pfimbi must not throttle replication. However, since Pfimbi monitors only the low-level disk latency, it waits until the buffer cache has been drained and thereby wastes time that could

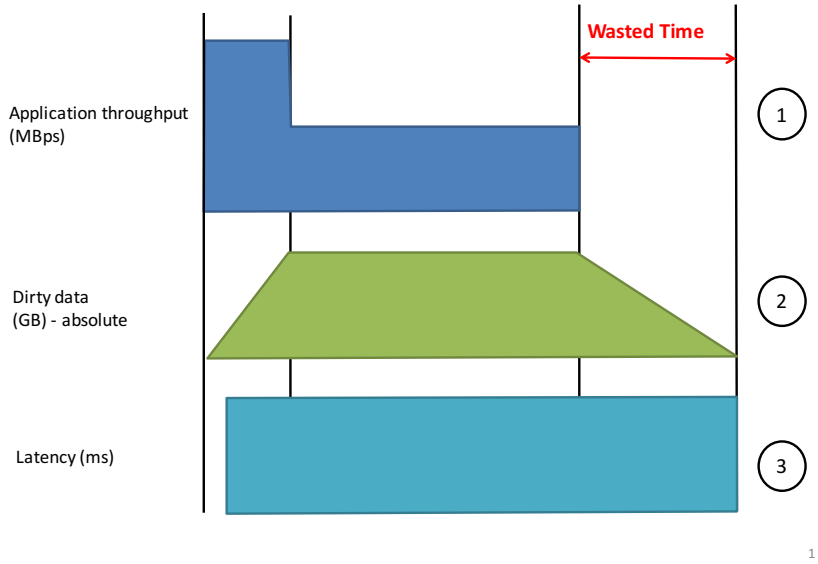


Figure 4.2 : Resource waste due to use low level metric for activity

have been used for replication. Figure 4.2 show the period of wasted time when the buffer cache is being drained.

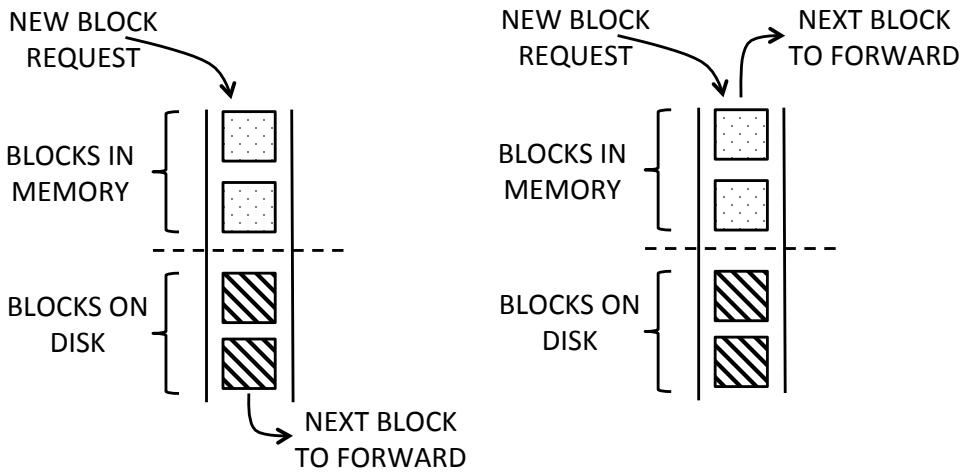
To avoid this waste, we needed a metric that captures activity from the userspace. One such metric is the sum of the disk device throughput and dirty data throughput. Dirty data throughput is the rate of change of data dirty. Note, this rate can be negative. We call this sum MemDisk throughput. In the waste scenario in Figure 4.2, MemDisk throughput will be zero, which signifies low activity; hence Pfimbi will not throttle replication. The bulk of our evaluation uses the latency metric, but Section 5.5 shows how using the MemDisk throughput improves resource utilization.

4.6.2 Forwarding blocks in LIFO order could reduce the overhead of background replication

In HDFS, only 5MB of data can be in flight in the pipeline and this data is always in memory. However, Pfimbi allows for much more data waiting to be forwarded at intermediate nodes. For explain if there are several gigabytes of data are waiting to be forwarded, Pfimbi can not force this data to remain in memory. Instead, Pfimbi simply reads the block files from disk when they are being forwarded. If the read is accessing cached data, then there is no penalty. However, when memory is low, reads may no longer come from the buffer cache, so Pfimbi incurs the penalty of extra reads from disk. Blocks read from disk increase the amount of work to be done in the system and hence the work span.

To totally eliminate extra reads, background replication should devolve to an in-memory pipeline when memory is low. A system that can perfectly switch between asynchronous and synchronous replication must be able to predict future cache usage perfectly. Once some blocks are replicated asynchronously, Pfimbi can no longer guarantee that these blocks will remain in memory until when they are forwarded. Solving this problem requires coordination between Pfimbi and the operating system's buffer cache management. However, without implementing such cross layer coordination, we can reduce extra reads by smartly choosing which blocks to forward at any given time.

Pfimbi by default stores blocks waiting to be forwarded using a queue in FIFO order. However, the oldest block is at the head of the queue so it will most certainly be read from disk when memory is low at illustrated in Figure 4.3(a). Instead of a queue as inFigure 4.3(a), we use a stack as in Figure 4.3(b). By choosing to receive the block most recently queued (LIFO), we have the highest chance of picking a block



(a) Queue implementation of Pfimbi (b) Stack implementation of Pfimbi

Figure 4.3 : A stack maximizes the chance of forwarding a block that is still cache in memory

that is still cached in memory.

4.7 Implementation

We now discuss the various implementation choices made in Pfimbi. To implement Pfimbi we augmented the HDFS DataNode with a module called Replication Manager (RM). In this section, we use RM to refer to the per-node component implementing the Pfimbi design.

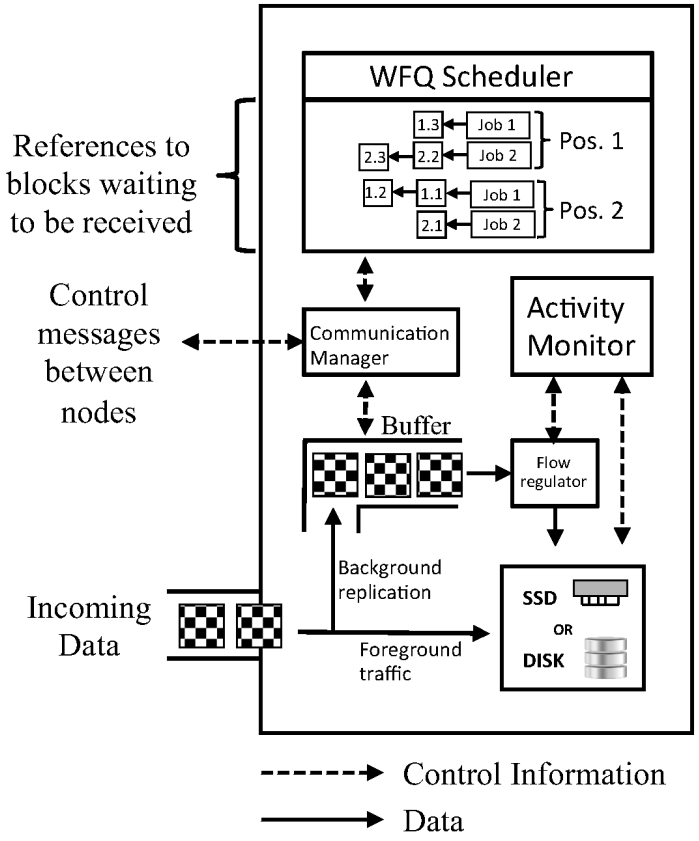


Figure 4.4 : Components of the Replication Manager in a Pfimbi node: A replication manager exists for each type of disk medium on a node.

Figure 4.4 presents the overall architecture of the Replication Manager. Pfimbi demultiplexes arriving traffic and directs background replication data to buffered in memory before writing it to disk. One component of the RM decides when to write the

buffered data to disk, based on disk activity. This component, which is labelled Flow Regulator in Figure 4.4, monitor throttles replication when the disk is busy. The disk activity information is collected and classified by the component labelled Activity Monitor. The Activity Monitor is in charge of getting disk activity measurements from the underlying OS as required (see Section 4.2.2). The Communication Manager implements the light weight protocol we use to initiate block transfers. When the buffer has free space, the Communication Manager asks other nodes for replication blocks. The WFQ Scheduler decides which node to receive data from next. The WFQ Scheduler stores information about replication blocks that will be received from other nodes.

Monitoring disk activity: The Activity Monitor regularly samples disk activity to determine whether disks are busy or not. To begin, Pfimbi monitors each disk separately and then computes for each disk type (SSD or HDD). Pfimbi computes an average because disks of the same type are picked in a round robin fashion to service incoming blocks.

The next step is to classify the obtained samples as either high or low activity. We adapt k-means [21] clustering to perform the classification. We set k=2 to obtain two clusters representing high and low activity. In Pfimbi, each cluster is described by the exponentially weighted moving average (EWMA) of all samples assigned to it so far. The EWMA will respond to changes in latency without Pfimbi having to reset the clusters regularly. Our threshold algorithm (shown in Algorithm 1) assigns each new incoming sample to the cluster closest to it. The threshold between high and low activity is the mean of the centers of the two clusters.

We can adjust the sensitivity of the EWMA can by changing the value of α.

Flow control: Between nodes, the RM controls the flow of background replica-

Algorithm 1 Threshold algorithm

```
1: high ← 1
2: low ← 0
3: while running() do
4:     curr ← currentActivity()
5:     if | curr − high | ≤ | curr − low | then
6:         high ← (1 − α) × high + α × curr
7:     else
8:         low ← (1 − α) × low + α × curr
9:     end if
10:    threshold ← (high + low) ÷ 2
11: end while
```

tion blocks to ensure that receiver side buffers, as shown in Figure 4.4, are always close to fully occupied. It is the job of the receiver to initiate asynchronous block transfers. However, receivers do not know that blocks are destined for them as they are not part of a synchronous pipeline for that block. The Communication Manager (see Figure 4.4) uses Remote Procedure Calls (RPCs) to let senders inform downstream nodes of blocks destined for them. We build on the existing HDFS RPC framework.

When a node **S** has an asynchronous block waiting to be forwarded to another node **D** downstream, node **S** sends an RPC containing: **S**, Block ID, Block Size, Flow Name, Flow Weight, and the position of node **D** in the pipeline. When node **D** decides to receive the block, it sends an RPC to node **S** to start block transfer. Node **D** only needs to send node **S** the Block ID. Node **S** then sends the block with the block ID. After node **D** receives the block, if there is any node downstream from it, node **D** will inform that node that the block is available.

Weighted fair queuing: In Pfimbi, upstream nodes inform downstream nodes of blocks waiting to be sent to them. Each node then keeps a record of the blocks it is yet to receive. Before initiating reception, the RM in a node must first decide which block to receive. To schedule these transfers, we implement an efficient approximation of weighted fair queuing suggested by [22].

Each block is associated with a queue corresponding to the source client. Pfimbi implements two level weighted fair queueing. In the first level, Pfimbi maintains a queue for each replica position, allowing earlier replicas to be assigned larger weight. In the second level, there is a queue for each job, enabling prioritization between jobs. Our WFQ-ing decides from which position, and which job to initiate the next block reception. When we receive a block notification, we create a reference to that block and insert this reference into the corresponding queue.

The algorithm maintains a system virtual time function $v^s(\cdot)$. When a block reference to the k-th block of queue i reaches the head of the queue, it is assigned a virtual start time s_i^k and a virtual finish time f_i^k. The algorithm then applies the earliest start time first policy to choose the block to initiate the transfer for. The system virtual time function is given by $v^s(\cdot) = (s_{min} + s_{max})/2$, where s_{min} and s_{max} are the minimum and maximum start times among all active head of queue blocks. This ensures that the discrepancy between the virtual times of any two active queues is bounded. Furthermore, $s_i^k = max(v^s, f_i^{k-1})$, and $f_i^k = s_i^k + l_i^k/w_i$, where l_i^k is the length of the block.

Chapter 5

Evaluation

In this section, we report experimental results to quantify the performance of Pfimbi.

Software setup: We implemented Pfimbi by extending Hadoop 2.2.0. Since we began this work, more recent Hadoop versions have been released but to our knowledge there have been no major changes to the pipeline replication mechanism [23]. Therefore, Pfimbi can be readily integrated into these releases. We compared Pfimbi against default Hadoop 2.2.0. We run Hadoop and Pfimbi on top of the YARN resource allocator. We also compared against setRep, a Hadoop mechanism that can be used to perform background replication. setRep increases the replication factor of a file, and it can be called immediately after a job finishes.

Hardware setup: We used 30 worker nodes and one master node. The worker nodes hosted HDFS DataNodes and YARN NodeManagers. Hence, computation was collocated with data storage. The master and worker nodes had the same hardware configuration. Each node had two 8-core AMD Opteron 6212 CPUs and a 10GbE connection. Nodes had 128GB of RAM, a 200GB SSD, and a 2TB hard disk drive. There were no other jobs running on the nodes during our experiments.

Configurations: We represent the tested configurations using the format DFS(#copies, #synchronous copies). Pfimbi(3,1) means using Pfimbi to write 3 copies, with only 1, the primary, being created synchronously. 2 replicas are created asynchronously. HDFS(3,3) means using HDFS to create 3 copies, all synchronously. For HDFS, the two numbers will always be the same since HDFS uses synchronous

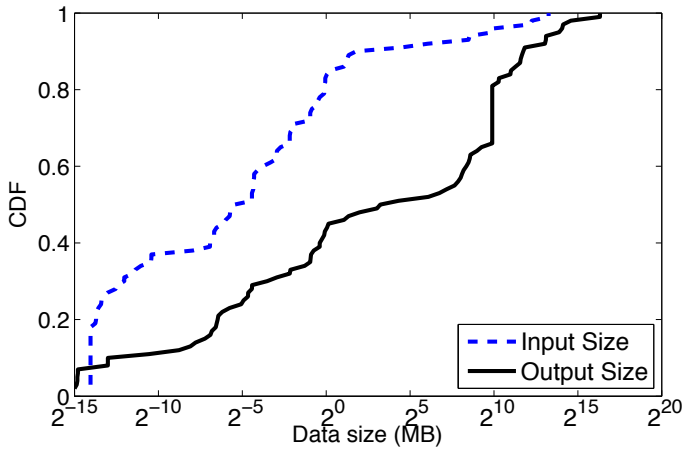


Figure 5.1 : CDF of job input and output sizes for the SWIM workload

replication. For setRep, we use setRep(3) to denote an increase of the replication factor from it's current value to 3.

Default Pfimbi parameters: The maximum size of the per-node buffer used for storing background replication blocks was 16. Replication flows used a flow weight of 1.0 unless otherwise specified, and by default, there was no active prioritization of replicas that are at earlier pipeline positions.

Metrics: We used four metrics to measure the performance of Pfimbi:

- Job runtime: the time it takes a job from the start time until it finishes all synchronous copies.
- the time from the start time until all first replicas are completed, giving resilience against any single node failure.
- the time to complete all replication work.
- the number of block writes per second.

Job runtime is a measure of job performance while the other metrics give information about the resilience and efficiency of the system. A job reports completion to the user after it finishes writing all synchronous copies. In HDFS, all copies are created

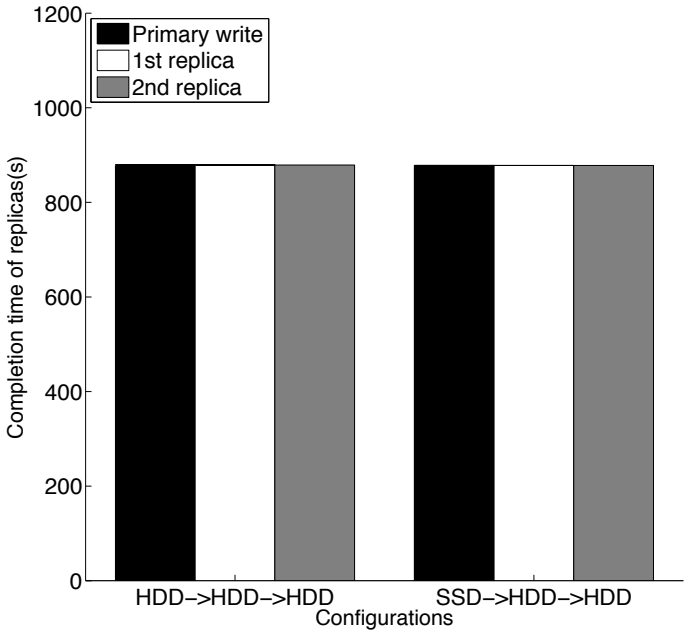
synchronously, so the time to write the primary will be identical to the time taken to write all the copies. However, when using Pfimbi, these metrics will be different which is why we consider them separately.

Workload: We used three workloads to test our system: a SWIM workload derived from a Facebook trace [24], Sort jobs, and DFSIO jobs [25].

SWIM [20, 26] is a MapReduce workload generator. Given an input trace, SWIM synthesizes a workload with similar characteristics (input, shuffle and output data size, job arrival pattern). The SWIM workload was derived from a 2009 Facebook trace [24] and contains the first 100 jobs generated by SWIM. Figure 5.1 illustrates the distribution of job input and output sizes in this workload. Most jobs were small and read and wrote little data while the few large jobs wrote most of the data. Such a heterogeneous workload is popular across multiple data center operators [15, 27]. Including replication, roughly 900GB of data is written into the DFS. In our experiments we gradually scaled up the workload by up to 16× (i.e., 14TB). This allowed us to observe the behavior of Pfimbi under light, moderate and heavy load conditions.

To analyze Pfimbi's benefits in a more controlled environment, we used Sort jobs. In all our Sort experiments, we sorted 1TB of randomly generated data. Sort experiments were repeated three times for each configuration.

Lastly, we used DFSIO [25] as a pure write intensive workload. DFSIO is routinely used to stress-test Hadoop clusters. Unlike Sort, DFSIO is storage I/O bound for its entire duration, and its only writes are to the DFS. This enabled us to analyze the behavior of DFS writes in isolation from the non-DFS disk writes (task spills, mapper writes) encountered in other Hadoop jobs.



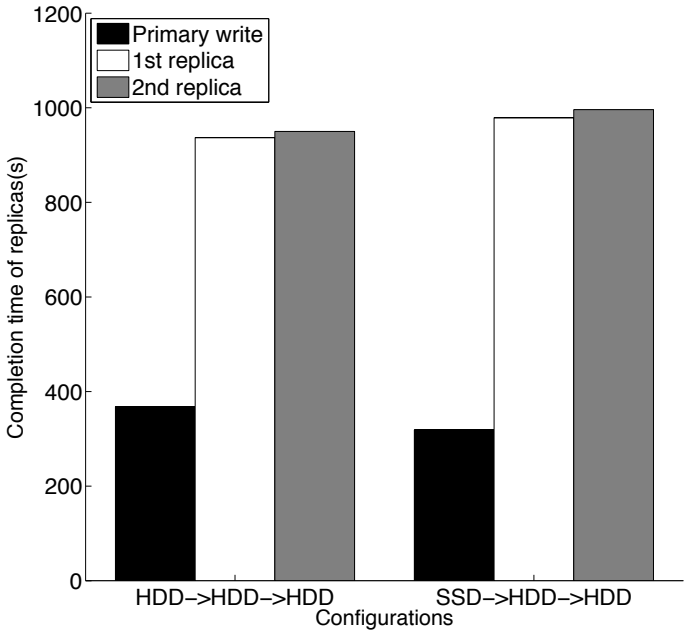
(a) HDFS(3,3)

Figure 5.2 : HDFS(3,3) cannot benefit from SSDs because of synchronous replication.

5.1 Pfimbi improves job runtime

We started by analyzing Pfimbi's benefits on an I/O intensive workload. We ran two DFSIO jobs back to back. We also analyzed the ability of HDFS and Pfimbi to leverage heterogeneous storage by varying the location of primary writes (SSD or HDD). The two replicas always went to HDDs. Thus, we had the SSD→HDD→HDD and HDD→HDD→HDD configurations. In the SSD case the primary writes did not conflict with replica writes, so for a fair comparison with the HDD case we partitioned the nodes into two groups. Half of the nodes handled the replicas while the other half handled the primary writes and the tasks. *Partitioning was only necessary for this experiment. For all our other experiments, we did not partition our nodes.*

Figure 5.3 illustrates the results for the two DFSIO jobs. Pfimbi significantly lowered the completion time for primary writes and completed the other replicas



(a) Pfimbi(3,1)

Figure 5.3 : Pfimbi finishes primary writes much faster due to the decoupled design and also benefits from switching to SSD.

with only a small penalty. HDFS(3,3) cannot benefit from moving the first copy write to SSD because it is using synchronous replication that is bottlenecked by the replica writes to the slower HDDs. With Pfimbi, the primary write duration improves when we move to SSD. The buffer cache absorbed a large proportion of the writes hence the improvement is not the same as the speed ratio between SSDs and HDDs.

We ran the SWIM workload to evaluate Pfimbi's benefits under different load conditions. We vary the load on the storage subsystem by scaling up the workload data size. Figure 5.4 shows the results. It plots the average job runtime under Pfimbi(3,1) normalized to the average job runtime under HDFS(3,3). Under a moderate workload (4x, 8x scaling) Pfimbi showed a 10-15% improvement in average job runtime. The per-job improvements (not illustrated) are up to 300% for small jobs (writing 1GB), and up to 30% for the most data-intensive jobs (writing 80GB). We also ran

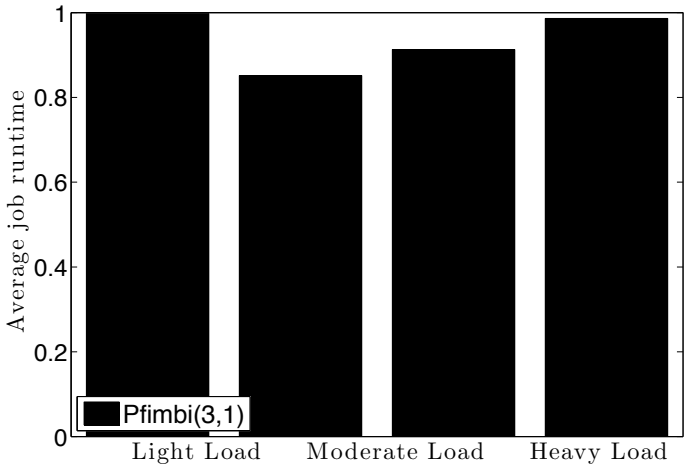
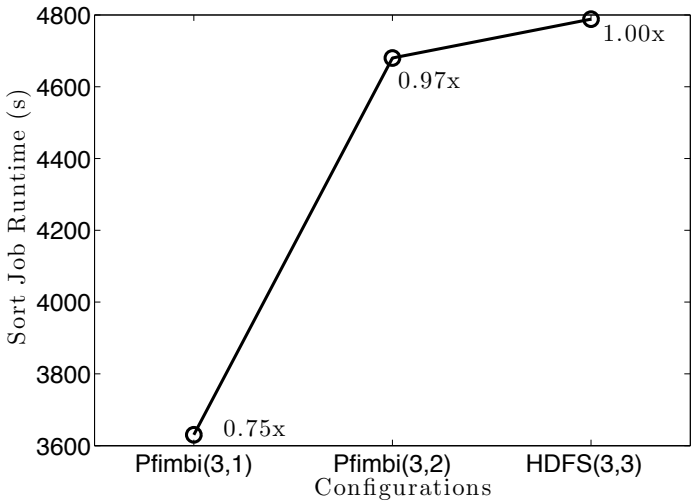


Figure 5.4 : Average SWIM job runtime with Pfimbi(3,1) normalized by average runtime with HDFS(3,3) under varying workload scaling factors. Pfimbi outperformed HDFS by up to 15% on average under moderate I/O load; performance gains for individual jobs can be much higher (see Section 5.1). For light load there was very little contention; at heavy load the system reached a saturation point, so it was increasingly difficult for Pfimbi to find idle disk bandwidth. In these cases, Pfimbi and HDFS perform similarly as expected.

a light workload (1x-2x scaling) and a heavy workload (16x scaling). Pfimbi did not show significant benefits for these extreme workloads. For the light workload, there was very little contention caused by replication in HDFS(3,3) while for the heavy workload the system was saturated so there was very little disk under-utilization for Pfimbi to leverage.

Pfimbi shows improvements also when we run the Sort job in isolation. Figure 5.5(a) shows that as we decreased the length of the synchronous portion of the pipeline, the job runtime decreased. The runtime of Pfimbi(3,1), which is purely asynchronous was 25% less than HDFS(3,3), which is purely synchronous. This improvement was because Pfimbi reduces contention between foreground and background traffic.



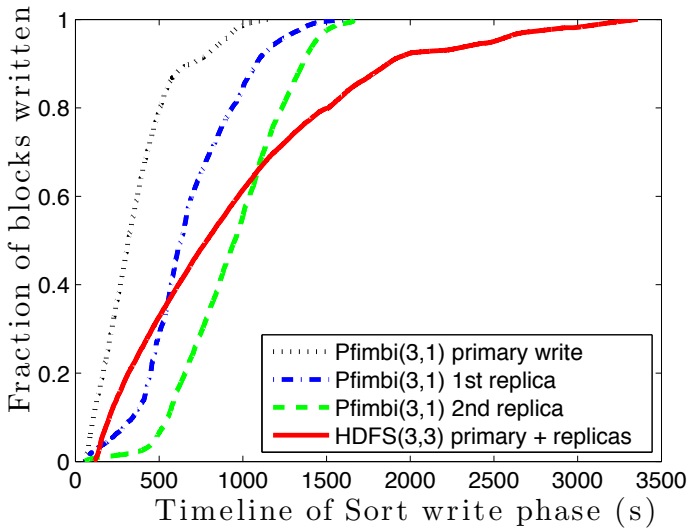
(a) Pfimbi improves Sort runtime.

Figure 5.5 : Running a Sort job under HDFS and Pfimbi

5.2 Pfimbi performs replication efficiently

We next analyzed how efficiently Pfimbi performs replication. We used the single Sort job to decouple the results from the influence of concurrent jobs. Figure 5.6(a) shows that replication in Pfimbi finished faster than in HDFS even when considering all the replicas. Pfimbi did not simply wait until the end of the job to start replication. Instead, Pfimbi filled in much of the replication into periods of low utilization at various nodes in the cluster.

For HDFS(3,3) blocks at all stages in the pipeline were created at the same rate since replication was synchronous. For Pfimbi, the blocks for the primary writes proceeded at a much faster rate because they did not have to contend with replication writes. As the number of primary block completions decreased, Pfimbi started creating more and more replicas.



(a) Pfimbi completed the writing of replicas at all positions completed faster

Figure 5.6 : Running a Sort job under HDFS and Pfimbi

5.3 Pfimbi protects primary writes from background replication

Asynchronous replication can also be done using the setRep mechanism in HDFS. After a file has been written, the setRep command was invoked to increase the replication factor of its blocks. Though asynchronous, setRep did not have any capabilities to manage and schedule background replication traffic and this results in contention with primary writes. In Figure 5.7, the setRep replication for the first job caused the second job to run much slower as compared to Pfimbi. Pfimbi was 33% faster. setRep can be tuned to be less aggressive, but this results in replication taking much longer than necessary.

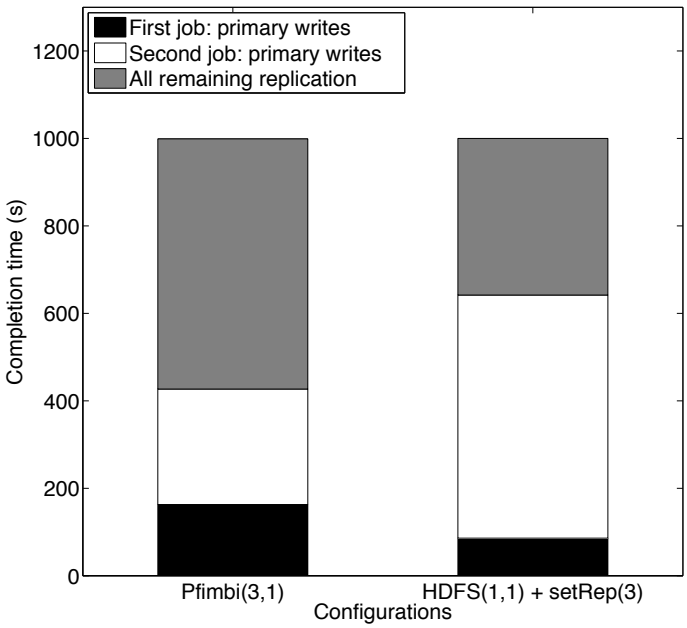


Figure 5.7 : Pfimbi vs. HDFS(1,1)+setRep(3) for two back-to-back DFSIO jobs. setRep was called immediately after a job was completed. Primary writes for the second job completed much faster in Pfimbi because they did not contend with background replication. In setRep(3) background replication still interfered with primary writes.

5.4 Pfimbi can divide bandwidth flexibly

Flexibly dividing disk bandwidth between jobs: We ran three concurrent DFSIO jobs and varied their flow weights. We launched the third job 500 seconds after the first two. Each job wrote 600GB of data. Figures 5.8(a) and 5.8(b) show the rate of block completions for the three jobs.

Figure 5.8(a) shows the experiments where the flow weights are equal, so the three jobs' replication writes shared the available bandwidth equally, giving similar block completion rates. Figure 5.8(b) illustrates the result when we use different weights for the jobs in the ratio 1:4:16. From 200-500 seconds, the ratio of Job 1 and 2's block completions matched the 1:4 ratio of the flow weights. Likewise, from 800-1000 seconds, the block completions of all three jobs reflected the assigned flow weights.

By assigning Job 3 a higher replication weight, its replication finished much sooner (Figure 5.8(b)) than when the jobs had equal weights (Figure 5.8(a)). The low block completions at the beginning of the experiment and after 600 seconds in Figures 5.8(a) and 5.8(b) were both due to replication being throttled for the primary writes of the jobs.

Prioritizing earlier pipeline positions within a job: Next, we demonstrated how prioritizing replicas that occur at earlier positions in their pipelines can help Pfimbi progressively achieve increasing levels of failure resilience. To better make this case we configured one DFSIO job to write four copies of its output data (3 replicas).

Figure 5.9 shows the number of block completions versus time for replicas at different positions in pipelines. Figure 5.9(a) illustrates we did not give priority to earlier positioned replicas, and we observed the 2^{nd} replicas and 3^{rd} replicas being created at the same time as the 1^{st} replicas. A user may prefer all of the 1^{st} replicas to be given priority. Figure 5.9(b) shows the case where we set Pfimbi to give strict priority to replicas at earlier positions in the pipelines when selecting the block to be received next. This reduced the overlap between the writes of blocks at different positions in the pipeline and replicas at earlier positions were finished sooner.

5.5 Using MemDisk throughput as an activity measure improves utilization

To evaluate the resource utilization of Pfimbi, we ran a simple workload, RandomWriter, which just wrote random data into the filesystem. The job wrote 400GB of data. The data had a replication factor of 3, meaning a total of 1.2TB of data were

written. At the time we ran this set of experiments, we had access to fewer servers, so we used only 10 DataNodes instead of the 30 we used in the earlier experiments. We measure the time taken for the primary writes to complete as well as the time for replicas to finish being created. Table 5.1 shows the amount of time taken to complete writing blocks at each position. Pfimbi results in an increase in the work span of about 35%. In these experiments, due to the data size, there were barely any extra reads, so the increase is due to lower resource utilization.

Scheme	**Primary (s)**	**1st replica (s)**	**2nd replica (s)**
HDFS	818	818	818
Pfimbi	255	1086	1109

Table 5.1 : Job completion time and workspan for RandomWriter job

Figure 5.10 shows the disk throughput at a single node in both HDFS and Pfimbi, for one iteration of RandomWriter. When using Pfimbi, there were several periods of low throughput. Using disk latency to measure whether the system is busy or not is the main cause in the drop in utilization. Changing our metric from disk latency to MemDisk throughput greatly reduced the work span.

Table 5.2 shows the improved duration. Pfimbi still resulted in a small increase in work span of 8%. With only an 8% increase in work span, we improved job duration by 70%. In future work, we hope to eliminate this waste.

5.6 Processing replicas in LIFO order reduces amount of extra reads

We also ran RandomWriter on a cluster with 10 DataNodes. However to force some blocks to be pushed out of the cache before they were forwarded, we blocked out

Scheme	Primary (s)	1st replica(s)	2nd replica (s)
HDFS	818	818	818
Pfimbi with new MemDisk thresholds	239	873	884

Table 5.2 : Using sum of memory and disk throughput greatly improves Pfimbi workspan

100GB of RAM, leaving only 28GB available. This enabled us to compare our stack and queue-based implementations. Table 5.3 shows a 15% reduction in work span simply by using a stack instead of a queue. We maintained a separate stack for each application to avoid starvation.

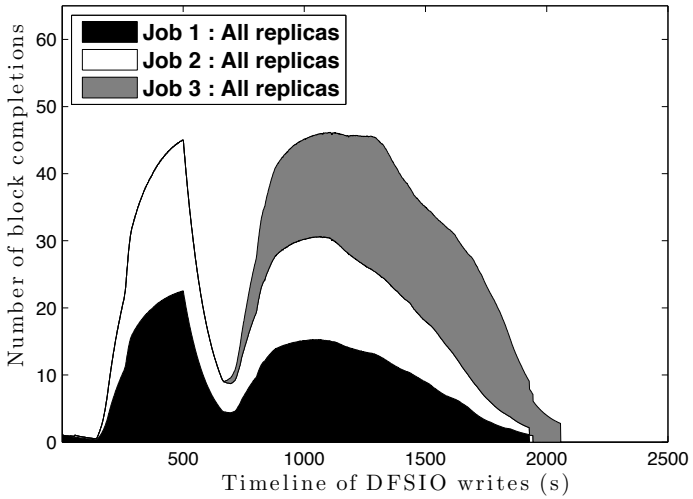
Scheme	Primary (s)	1st replica(s)	2nd replica (s)
HDFS	960	960	960
Pfimbi using a queue	651	1611	1621
Pfimbi using a stack	551	1371	1380

Table 5.3 : Using a stack reduces the amount of extra reads

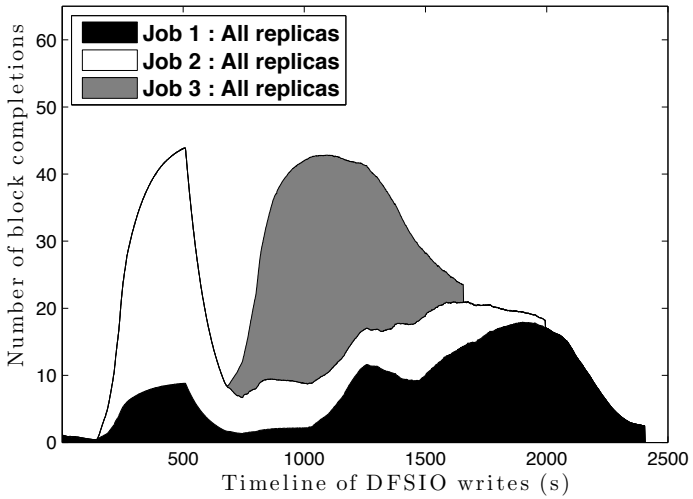
The improvement in work span was the result of more reads coming from blocks that were still cached in memory. Table 5.4 shows the total amount of data read from disk during the experiment.

Scheme	Disk read total (GB)
HDFS	0.2
Pfimbi using a queue	490.8
Pfimbi using a stack	370.4

Table 5.4 : Using a stack reduces the amount of extra reads

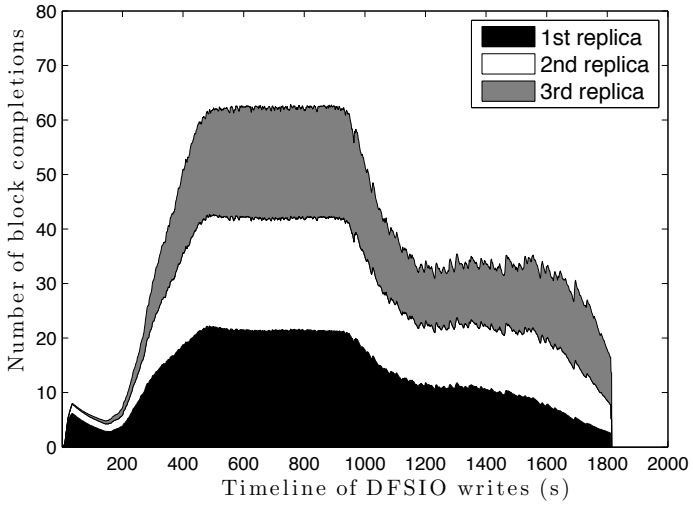


(a) Flow weights ratio 1:1:1

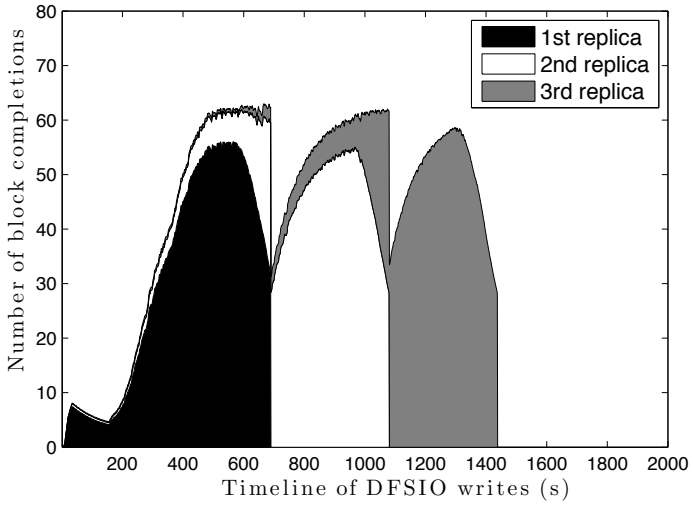


(b) Flow weights ratio 1:4:16

Figure 5.8 : Three DFSIO jobs ran concurrently with different replication flow weights. Each job had a replication factor of 3. Job 3 was started 500 seconds later than the first two jobs. (a) Flow weights were equal. (b) When flow weights were different we observed bandwidth sharing at proportions according to the ratio of the flow weights, thus Job 3 was able to achieve failure resilience much sooner.

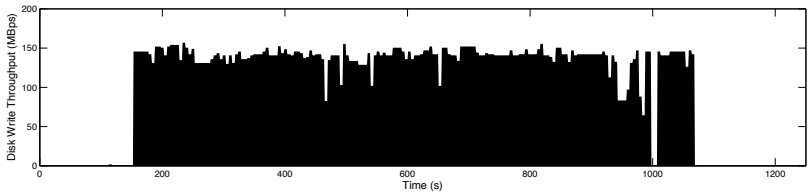


(a) Fair sharing

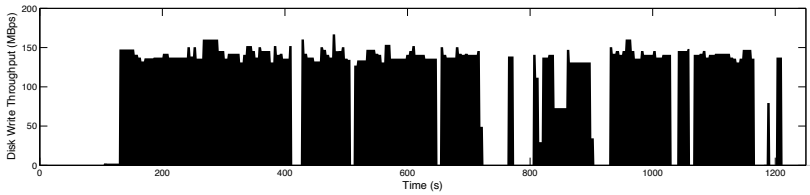


(b) Prioritizing earlier replicas

Figure 5.9 : A single DFSIO job with replication factor 4 under Pfimbi. (a) The three replicas shared the available bandwidth fairly at a 1:1:1 ratio. (b) The ratio was set to 100:10:1. This resulted in earlier replicas finishing sooner, achieving progressively increasing levels of failure resilience. Even if a failure were to occur at 800s, the data would still be preserved.



(a) Throughput in HDFS is consistently high



(b) Pfimbi has several periods of low throughput

Figure 5.10 : Disk throughput for HDFS and Pfimbi during a RandomWriter execution

Chapter 6

Conclusions and Future Work

6.1 Conclusions

HDFS is often regarded as being "good enough," a sentiment with which we disagree. We have systematically presented the limitations of synchronous pipeline replication and the case for flow-controlled background replication as a natural next step in the design of distributed file systems for big data ecosystems. Our proposal, Pfimbi, represents merely a specific point in the design space,yet it already shows that flow-controlled background replication is a highly viable approach with numerous benefits. Given the foundational role that DFSs plays in big data ecosystems, these benefits will likely lead to large impact.

In this work, we first employee asynchronous replication, a well known technique, to decouple primary writes from replication work. Asynchronous replication protects applications from bottlenecks that may exists in write pipelines. However, primary writes will still contend for limited disk bandwidth with replication writes. To isolate these two data streams, Pfimbi controls the flow of replication traffic. Pfimbi is able to prioritize primary writes while maintaining high utility.

6.2 Future work

6.2.1 Automated asynchronous replication

Although Pfimbi provides mechanisms for flow-controlled asynchronous replication, the user still decides when to use them. This decision is non-trivial. For example when the IO system is heavily utilized and there is no idleness, replication should be performed synchronously. Additionally, asynchronous replication may introduce extra reads from the disk if the amount of pending replication data grows too large. Before this happens, the user should revert back to synchronous replication. The next step in developing Pfimbi it to enable it to make these decisions automatically. This would involve more monitoring and predicting future load.

6.2.2 Analysis of fault tolerance

Another direction for future work is an analysis of the implications of Pfimbi on fault tolerance. The mechanisms required to recover from failures in Pfimbi are the same as those used in existing systems. However, Pfimbi changes when certain mechanisms may need to be invoked. In scenarios when Pfimbi delays replication, we ought to analyze whether the improvement in application performance outweights the potential cost of recovering data in the case of a failure. This analysis can be done automatically, but requires the filesystem to become aware of the provenance of data, that is, the lineage and duration of tasks that produced the data. Pfimbi could use this analysis to make online decisions whether to throttle replication traffic or not.

6.2.3 Task driven replication

In addition to providing resilience against failures, data replication improves the locality of data for future tasks. When replicating data asynchronously, we believe preference should be given to data that will be read soon. The amount of slack between the production of data and the use of data should be another input into Pfimbi's decision-making on what to replicate first and how quickly to replica it.

Bibliography

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[2] "Apache hadoop." `http://hadoop.apache.org/`, 2015.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10, 2010.

[4] "Archival Storage, SSD & Memory." http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html.

[5] "Memory Storage Support in HDFS." http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/MemoryStorage.html.

[6] D. Fetterly, M. Haridasan, M. Isard, and S. Sundararaman, "Tidyfs: A simple and small distributed file system.," in *USENIX annual technical conference*, 2011.

[7] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*.

[8] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pp. 231–242, ACM, 2013.

[9] "Ability for hdfs client to write replicas in parallel." `https://issues.apache.org/jira/browse/HDFS-1783`, 2015.

[10] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," in *Proceedings of the 2014 ACM conference on SIGCOMM*, pp. 331–342, ACM, 2014.

[11] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, (New York, NY, USA), pp. 6:1–6:15, ACM, 2014.

[12] F. Dinu and T. S. E. Ng, "Rcmp: Enabling efficient recomputation based failure resilience for big data analytics," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, (Washington, DC, USA), pp. 962–971, IEEE Computer Society, 2014.

[13] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat, "Themis: An i/o efficient mapreduce," in *SOCC 2010*.

[14] "Failure Rates in Google Data Centers." http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/.

[15] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big

data systems: A cross-industry study of mapreduce workloads," in *Proc. VLDB*, 2012.

[16] "Terasort." https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html.

[17] "Nutch." https://wiki.apache.org/nutch/NutchTutorial.

[18] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 41–51, IEEE, 2010.

[19] "k-Means clustering - basics." https://mahout.apache.org/users/clustering/k-means-clustering.html.

[20] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[21] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, pp. 281–297, Oakland, CA, USA., 1967.

[22] I. Stoica, H. Zhang, and T. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 2, pp. 185–199, 2000.

[23] "Hadoop releases." `http://hadoop.apache.org/releases.html`, 2015.

[24] "SWIM Workloads repository." https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[25] "Running DFSIO mapreduce benchmark test." https://support.pivotal.io/hc/en-us/articles/ 200864057-Running-DFSIO-mapreduce-benchmark-test.

[26] "SWIM." https://github.com/SWIMProjectUCB/ SWIM/wiki.

[27] "J. Wilkes - More Google cluster data." http://googleresearch.blogspot.ch/2011/11/more-google-cluster-data.html.